Filtered Ranked Segment Search for Visual Exploration in a modern Time Series Database

Brown CS2270 Project Summary
Mary McGrath and Junjay Tan
May 6, 2019

Table of Contents

Summary

Our project aims to support range-filtered time series segment searches in a modern time series database and to analyze bottlenecks and implement a UI/Database design solution that goes beyond Metro-Viz. Through our own work on anomaly detection and through conversations with another Brown lab, we found that being able to hone in on time series segments that met certain range criteria on processed outputs (anomaly detector likelihoods, wavelet power spectrums) is useful for visual exploration of interesting regions.

The main contributions of this project are as follows: (1) A UI built in React and Express that supports filtering, ranking, and previewing of segments by detector output value as well as panning/zooming across large time series data sets (at least 10M-100M+ rows) stored in a database, (2) an implementation of this functionality as a set of operators installed via an extension to a modern time series database based on PostgreSQL, TimescaleDB, (3) benchmarks showing our implementation on various sizes of data, and finally (4) a method for generating a large scale ECG time series dataset with anomalies based on an existing generator called ECGSYN. We also present initial work on an additional operator for auto-downsampled windowing operators that we believe would be useful for visual exploration of time series data by limiting data sent from the database to the UI when it drastically exceeds the screen resolution.

We conclude with some database and UI optimization opportunities along with features that could be added.

Motivation and Goals

Visually querying through a large time series data set to find interesting patterns or patterns at the "edge" of an anomaly detector threshold is not practical to do manually. Instead, an analyst needs a way to rank these patterns and select the best ones to do a "deep dive" on. Further, perusing these as individual points is repetitive and not representative of actual patterns; it is more accurate to think of and view these as continuous time *segments*.

We assume there is some processed time series output that is good to use for filtering interesting segments to analyze further, although our method can be used on the raw signal as well. Two specific use cases we've discovered are for anomaly detection and for exploring neural recordings related to long-term deep brain stimulation:

- Anomaly detectors are often based on some threshold value, which can be thought of as a processed value that can be represented as another time-series. This detector could be as simple as the moving average detector, or it could be more complex, such as a machine learning (ML)-based one. In the latter case (e.g., Greenhouse, LSTM-AD, Numenta HTM), each time point is associated with a ML output value, such as likelihood. This output is then compared to some threshold that is used to identify anomalies. However, analyzing values at the "boundaries", meaning some range below or above the threshold, is useful for understanding what an anomaly detector may be flagging incorrectly.
- The Brown Neuromotion Laboratory is at the beginning stages of receiving large amounts of deep brain stimulation data collected by devices installed in patients. Such devices have been used for many years in patients to ameliorate the effects of Parkinson's, but recently researchers have begun collecting from them. This data is different than existing neurological data because it is collected outside the lab, meaning annotations are not associated with time spans (normally lab techs would record that task X is occurring from time 1 to time 10) and because there are very large amounts of data. Further, scientists must explore the data because they are not sure what they are looking for. Our conversations with a researcher in the lab identified that a typical initial exploration step is performing wavelet spectrogram processing of the data so that each time point is associated with signal powers for each frequency. After this initial batch processing step is performed, the outputs are stored and the researcher starts visually exploring the data in Python or Matlab to try to find interesting segments, such as places where the power of a certain frequency is much higher than the total average signal power. This process is time consuming even for the smaller amounts of lab data they currently have, so they are unsure how they will be able to explore large volumes of deep brain stimulation data.

After applying a ranged segment filter, a user could prioritize analyzing windows around an uncertainty band by metrics such as recency, length of anomaly, amplitude, etc to get a list of **top-*k* segments**.

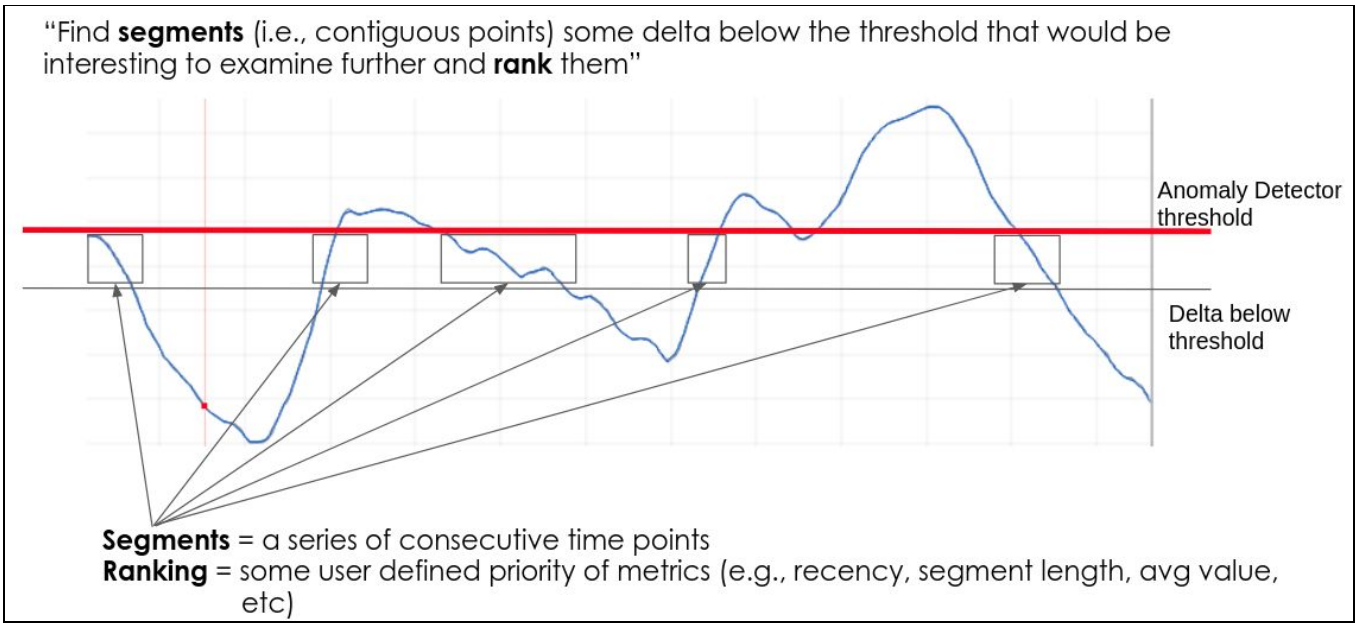

Figure 1: Example of segments within some filtered value range

Related Work

Our prototype builds on concepts discussed in Metro-Viz [Eichmann et al., 2019], namely visual exploration of time series data, but extends it to support a specific type of directed search (filtered segments meeting some range criteria) rather than relying on the user to manually explore the data. Additionally, it adds two user interactions, pan and zoom, uses a large time series data set for evaluation (~100M data points of synthetic ECG data) and relies on a modern time series database to store and query this data.

Various projects have focused on interactive querying of time series data, including TimeSearcher [Hochheiser et al, 2004] and most recently Kyrix [Tao et al., 2019]. TimeSearcher is very old at this point and focused mainly on interactive formulation of queries with smaller data sizes, at most tens of thousands of data points. Kyrix is particularly interesting because recent efforts have focused on pan and zoom of time series data containing 100M points at interactive latencies (<500ms). Kyrix relies on optimizations in the middleware layer and smart database design choices (without modifications to the databases themselves), and supports a few standard databases such as PostgreSQL and SciDB. It relies largely on R-tree indexes and lookup tables materialized in the database, along with caching and incremental view maintenance in the middleware/frontend layers. The paper came out very recently, so if we had more time, it would have been instructive to examine and adapt its ideas for pan and zoom in our project. However, it has no notion of segment-based filtering.

There has been a large amount of literature on top-K queries for relational databases ([Ilyas et al, 2008] provides a good survey) that rank a set of database objects based on various criteria. [Li et al., 2010] extends this work to temporal data, which ranks top-k objects at some time instance *t* using a SEB-tree, which is based on a B-tree. [Gao et al, 2018] extends Li et al.'s work to empass durable temporal queries, which ranks objects based on whether they have been in the top-k for some fraction of time in the past. We were curious how their work applied to ranking time series segments within some filter band, and discovered they were not completely applicable and more parallel and complementary.

Large Dataset Generation

We want to evaluate our system on complex large time series datasets. A large dataset should at a minimum contain millions of data points and be several GB in size, making it a non-trivial amount to send over the network from server to client and to process in a web browser. Future evaluations should probably use TBs of data to ensure data can't be cached completely in the server memory, but we use several GBs in our study as a good starting point.

Surprisingly, we could not find publicly available large time series datasets. The vast majority of time series datasets have at most tens of thousands of data points [Numenta Anomaly Benchmark, Keogh 2005]. The largest publicly available time series dataset we found was MIMIC, which contains 330M rows of a data in its *CHARTEVENTS* table [Johnson 2016]. However, this table contains data from many patients across a multitude of dimensions, so analyzing a single patient across a single dimension for anomalies would leave a much smaller dataset for evaluation.

Failing this, we began looking for dataset generators. We eventually found an ECG data generator called ECGSYN that could generate 1GB+ datasets containing 33M+ data points [McSharry 2003]. However, this generator segfaults after about 33M points (presumably because it performs all computations in memory), so we ran multiple passes of it to create several gigabytes of data, which we then stitched together. The seams contain discontinuities, but this is fine for the purposes of our study.

ECGSYN generated realistic looking ECG data with noise, but it did not add timestamps, anomalies to the data, or anomaly detector output. Regarding anomalies, we tried several methods to generate anomalies, including augmenting with pseudo-random values and adding sinusoidal functions, but neither of these looked realistic. In the end, we found that randomly creating segment overlaps, meaning superimposing an offset region of the dataset with itself at randomized regions in randomized lengths, created interesting looking anomalies.

Regarding generating detector outputs, we attempted to run LSTM-AD and greenhouse on the datasets, but it was non-trivial to extend those methods to tens of millions of data points. Instead, we settled on generating synthetic detector outputs, where the baseline output is a scaled moving average on the raw data, and anomalous regions have output values that are increased by some randomized percentage (140% to 200%) of the baseline value. Code for generating anomalies and detector outputs is in the repository. An example section of the generated ECG data, detector output, and anomalous ranges is shown in the following figure.

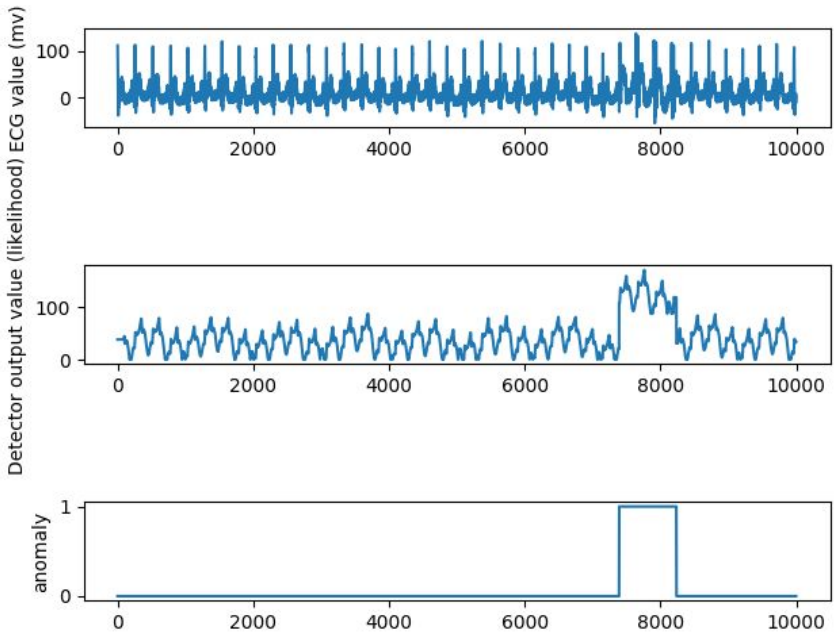

Figure 2: Synthetic ECG data (top), detector output (middle), and anomalous ranges (bottom) for a small segment of data

User Interface Design and Implementation

Design Rationale

The current Metro-Viz UI doesn't support the interactions needed in segment search. Phillip suggested we rebuild the UI in React, so our implementation uses a completely new front end and middleware layer built entirely in JavaScript and Node, with the frontend using React and Vega/D3 for visualization. The middleware layer communicates between the frontend and the database as shown in the following figure.

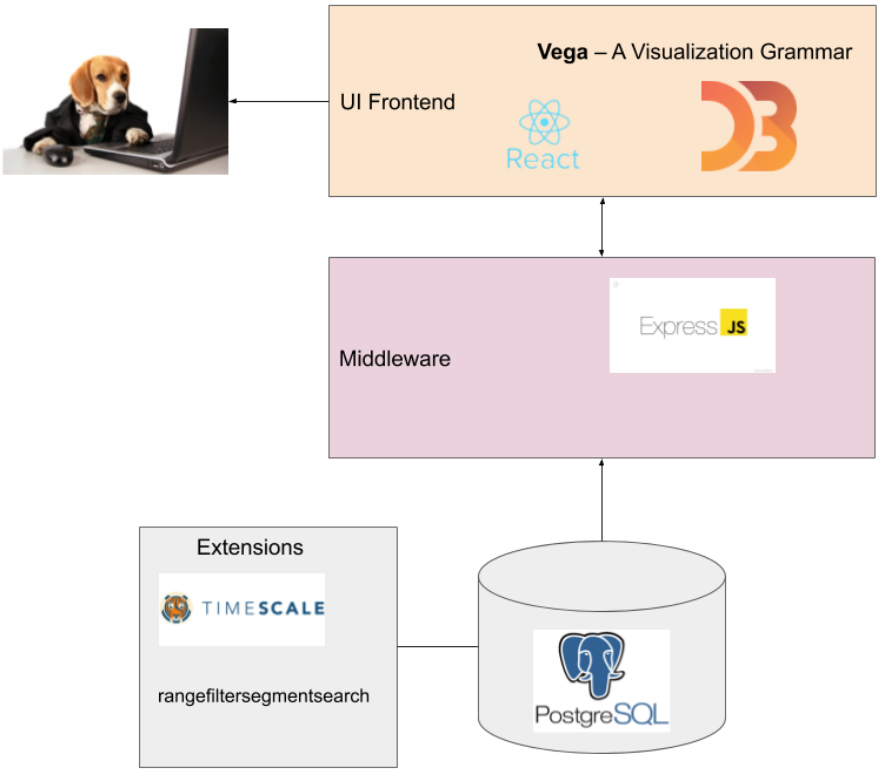

Figure 3: Architecture

Implementation

React was chosen for the front end framework due to its performance and composability. React only renders/re-renders components when a state or property change is detected, thus performing the minimum amount of computation needed. React was used in concert with Redux and Sagas to manage asynchronous state changes and fetching of data.

Vega is a grammar of interactive graphics developed by the University of Washington [Satyanarayan et al. 2017]. It builds on, and often compiles to, D3, but adds an abstraction layer for portability as well as a DAG implementation which asynchronously and iteratively builds a graphic, allowing for updating data and other signals without a full refresh of the visualization.

The original Metro-Viz UI loaded the entire dataset into memory, which is infeasible for large datasets. Using Vega allowed a reactive implementation of pan and zoom data exploration. We were able to add listeners to the x axis signals, then evaluate whether new data should be requested based on the view's closeness to the beginning or end of the loaded segment of the dataset. As new data is loaded into Vega, old data is also removed, allowing limitless panning and zooming.

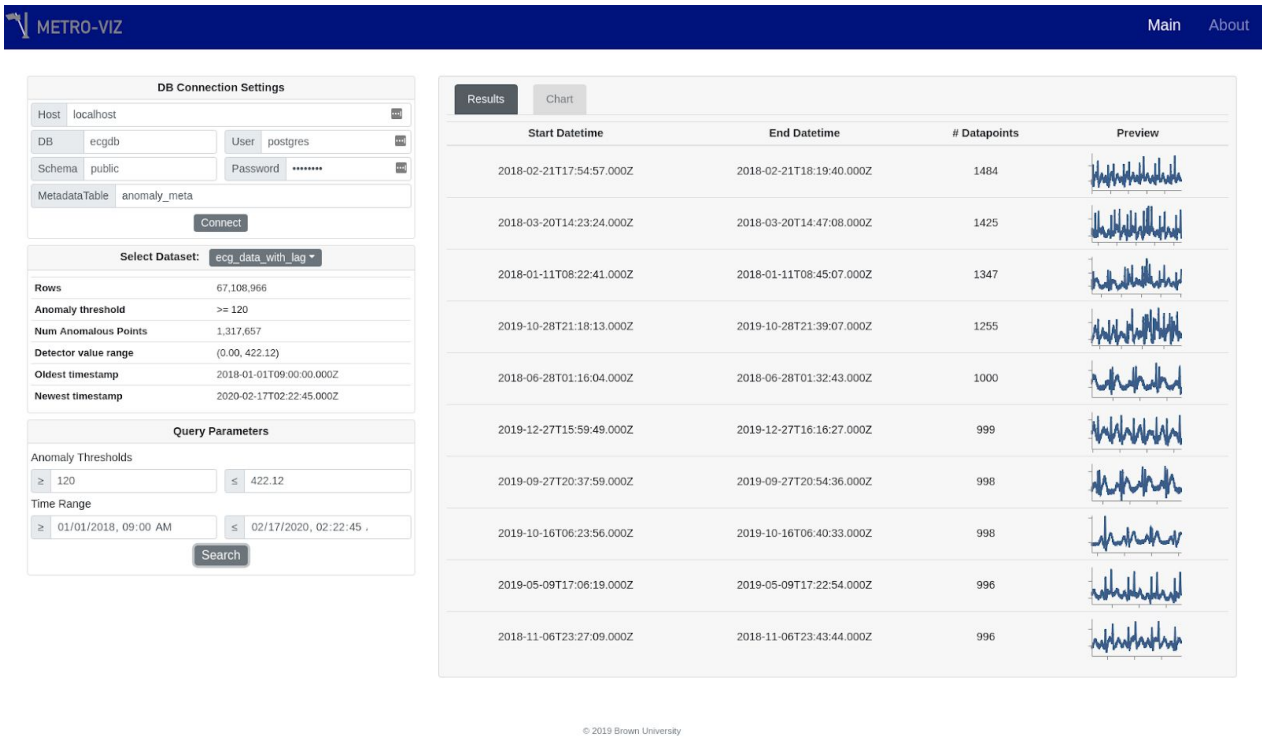

Figure 4: Screenshot of Segment Search Results and Preview

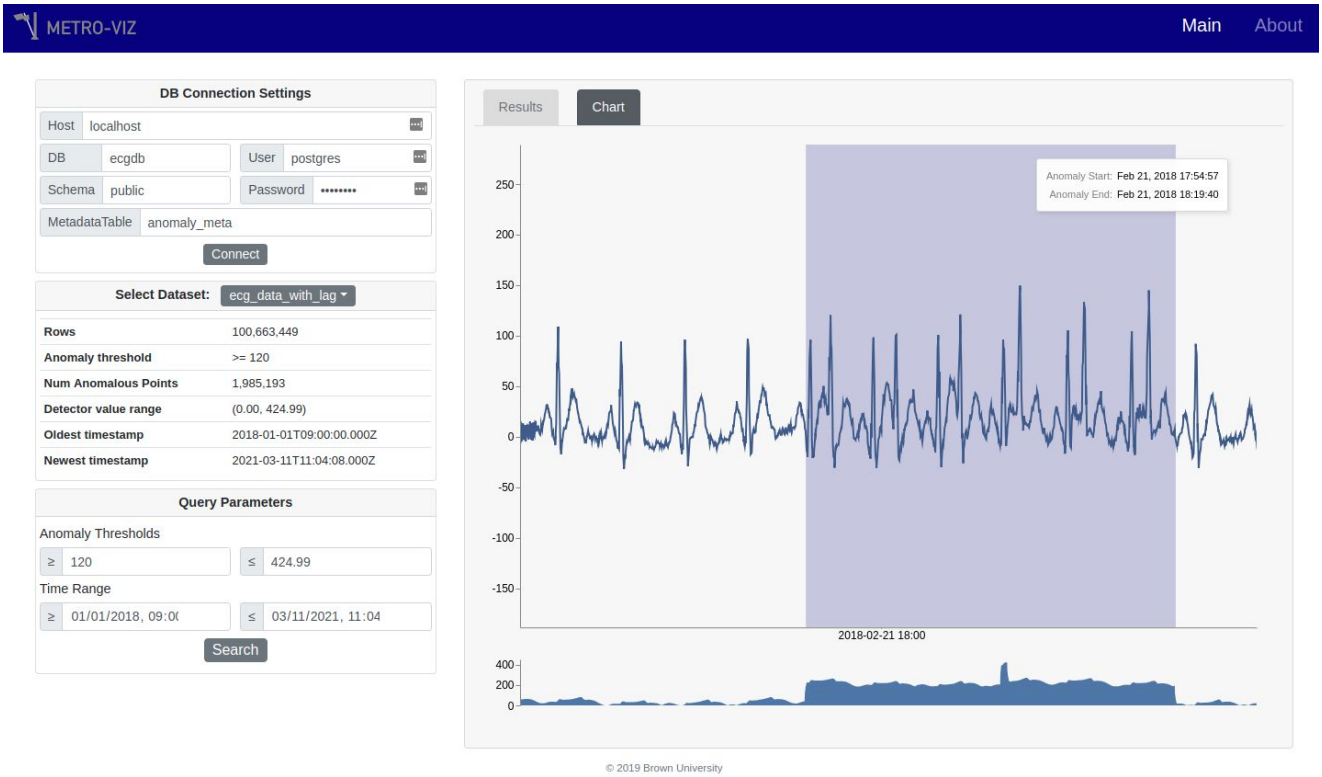

Figure 5: Screenshot of Drill Down View with Pan and Zoom

Note that the preview and main time-series charts account for non-uniform sampling of time points and plots accordingly.

Besides supporting segment filter search and pan/zoom, the interface also provides basic error messages if connections or queries fail. Future work could involve adding the following features:

- Add widgets to allow prioritizing by other metrics
- Add custom user functions (for prioritizing segments returned)
- Overlay anomalies, compare multiple detectors.
- Adapting visualization and search for multi-channel time series data
- Hardening implementation (error handling, security) for production ready release

Database Implementation (TimescaleDB) and Performance

Why We Selected TimescaleDB

Since we didn't want to build a database from scratch and instead wanted to build an extension that could leverage an existing system, we evaluated several modern time series databases: TimescaleDB, InfluxDB, and OpenTSDB. Our main selection criteria were that it be open source, extensible, and appear well written.

We narrowed our focus to TimescaleDB and InfluxDB because they were newer, and it appeared that adoption was picking up on these systems and moving away from OpenTSDB. We then selected **TimescaleDB** rather than InfluxDB for several reasons. Most importantly, InfluxDB is written from scratch in Go, uses custom operators, and uses a NoSQL key-value tag data model. These design decisions made it unclear how extensible it would be. Further, InfluxDB has recently moved to an enterprise-focused model that de-prioritizes open source, so it is unclear how much of its code is still open source anymore. In contrast, TimescaleDB is entirely open source and based on Postgres, making it clearly extensible. Its main selling point is an abstraction layer on top of postgres (via an extension) catered to the needs of time series data (append heavy, chunking by time, etc). More details about its architecture can be found in its documentation.

TimescaleDB claims much higher ingest rates than stock postgres.

Figure 6: Ingestion Time (s) for batches of 33.5M records on TimescaleDB vs Stock PostgreSQL

The downsides to TimescaleDB are that it is row store, does not compress/uncompress data on the fly to limit data transfer like modern OLAP databases, and is currently limited to single node. InfluxDB supports multi-node NOSQL style sharding; however, its community edition (which is also the part that is open source) is limited to a single node.

Database Schema

Besides the ECG data that was evaluated at sizes of 33.5M records, 67M records, and 100M records (1.1GB, 2.2GB, and 3.3GB uncompressed, respectively), we also used a smaller sine waves dataset, where the raw signal was represented as a sine wave and the detector output value was represented as another offset sine wave with a different amplitude and frequency (500M records, 25MB uncompressed). The ecg data table also had a second version that included an additional column containing each data point's previous timestamp, which allows much faster segment filter searches for reasons discussed in the following section. The table definitions are listed below.

```
CREATE TABLE IF NOT EXISTS "ecg_data"(
   ecg_datetime TIMESTAMP WITHOUT TIME ZONE NOT NULL,
   ecg_mv NUMERIC NOT NULL,  --ecg reading in millivolts
   anomaly_likelihood NUMERIC
);

CREATE TABLE IF NOT EXISTS "ecg_data_with_lag"(
   ecg_datetime TIMESTAMP WITHOUT TIME ZONE NOT NULL,
   ecg_datetime_prev TIMESTAMP WITHOUT TIME ZONE,  -- can be null
   ecg_mv NUMERIC NOT NULL,  --ecg reading in millivolts
   anomaly_likelihood NUMERIC
);

CREATE TABLE IF NOT EXISTS "sinewaves_data"(
   my_datetime TIMESTAMP WITHOUT TIME ZONE NOT NULL,
   sine1_value NUMERIC,  -- use this as the detector output
   sine2_value NUMERIC
);
```

TimescaleDB used each table's timestamp column for the hypertable definition, thereby chunking each hypertable by that date field. Additionally, B-tree indexes were added on the raw signal and the detector output fields. The latter is critical for allowing fast filtering.

Range-Based Segment Filtering

The main challenge in segment-based filtering is that one must keep track of sequences to know if sequential time points after performing a filter are indeed consecutive in time on the original data set.

Otherwise, two points could be adjacent in time after the filter, but not actually part of a contiguous segment in the full data set because there was a point in between that was filtered out. We assume that any data points not in the full data set do not exist, and therefore there is no way to know if there are missing data points between them in ground truth/golden data. I.e., all we see is all we know.

We explored several approaches to keeping track of sequence order. One approach is to generate a sequential index for each time value. However, this can pose a problem if a new data point arrives in time later for which many later points have already been stored, as we would then need to re-index all these later points. Instead, we focused on using the captured timestamps and calculating on the fly or materializing each time point's previous timestamp. In this way, if 10 points have arrived at time (t1, t2, …, t10), and a new point arrives later that had timestamp t0, only the time point at t1 would need to be updated so its previous timestamp value is now t0, with the t0 point being updated so its previous timestamp points to time t[-1].

We explored evaluating these previous timestamps "on the fly" using PostgreSQL's window + lag functions, as well as materializing them in an additional table column. These are implemented as a series of overloaded functions in a PostgreSQL extension in PL/pgSQL (SQL procedural language) that return a filtered relation containing an additional column that identifies the unique segment start time:

- filter_segments(TABLENAME, datecol TEXT, valuecol_to_filter TEXT, value_to_passthru TEXT, min_value NUMERIC, max_value NUMERIC)
- filter_segments(TABLENAME, datecol TEXT, prev_datecol TEXT, valuecol_to_filter TEXT, valuecol_to_passthru TEXT, min_value NUMERIC, max_value NUMERIC)

The general approach is to calculate the previous timestamp value for each entry in the full data set (or skip this step if this is already materialized in an additional column), then filter the results based on detector output values and date ranges, then order these results and finally loop through them to determine which points are adjacent in time to each other and hence in the same segment as summarized below.

```
If previous timestamp column is not provided:
    Determine previous timestamps
Filter by date and detector output range criteria
Order results by timestamp

# TS of the previous entry evaluated in the loop
loop_prev_ts = null

Loop through results:
 Each entry has a (cur_ts, prev_ts)
  If prev_timestamp is null:
    Segment_start_ts = cur_ts
    cur_prev_ts = cur_ts
  Else:
    If loop_prev_ts is null:
      Segment_start_ts = cur_ts
      Loop_prev_ts = cur_ts
    Else if prev_ts = loop_prev_ts:
      # Is a continuation of segment
      Loop_previous_ts = cur_ts
    Else:
      # Is a new segment
```

```
    Segment_start_ts = cur_ts
    Loop_previous_ts = cur_ts
Return tuple with segment_start_ts appended
```

Algorithm 1: Pseudocode for filter_segments

Note that both functions assume that timestamps are unique in the input relation; if you have duplicate timestamps then the previous timestamp values will not be evaluated correctly since one of the duplicates will have a previous timestamp value of itself. Forcing previous timestamps to be distinct can address this issue, but we did not use it because it would incur extra overhead.

We evaluated both approaches on a 4 core i5 machine running a very fast NVMe drive on a 33M data point (1.1GB uncompressed) synthetic ECG data set. Running on a TimescaleDB hypertable, we measured that a filtered segment search runs in ~18s in the variant where previous timestamps are calculated on the fly. Running on stock postgres tables showed a slightly higher runtime, at around 20s. Analyzing the EXPLAIN ANALYZE output showed that the vast majority of this time (65%) is spent determining the previous timestamps. In contrast, pre-materializing the previous timestamps simplifies the processing greatly as we do not have to do this on the full dataset and can simply filter them results and then join segments, bringing the query down to about 1s, a 16X improvement. Therefore, pre-materializing the previous timestamps is the preferred approach, and in a real-world deployment this column would need to be updated in batch at various times, similar in spirit to OLAP write/read stores.

Query performance measurements of segment filter search are presented in the following figure for 3 different table sizes. These were performed on a single server running an 8 core AMD FX-8320E processor with 20GB of RAM and a 1TB 7200 Western Digital HDD, on PostgreSQL 10. Cold cache measurements, in which the database was restarted and OS caches were cleared, along with warm cache measurements, in which SELECT COUNT(*) was run on each data table before each query, are presented for comparison. These searches are based on finding all anomalous segments, as in the query after the figure.

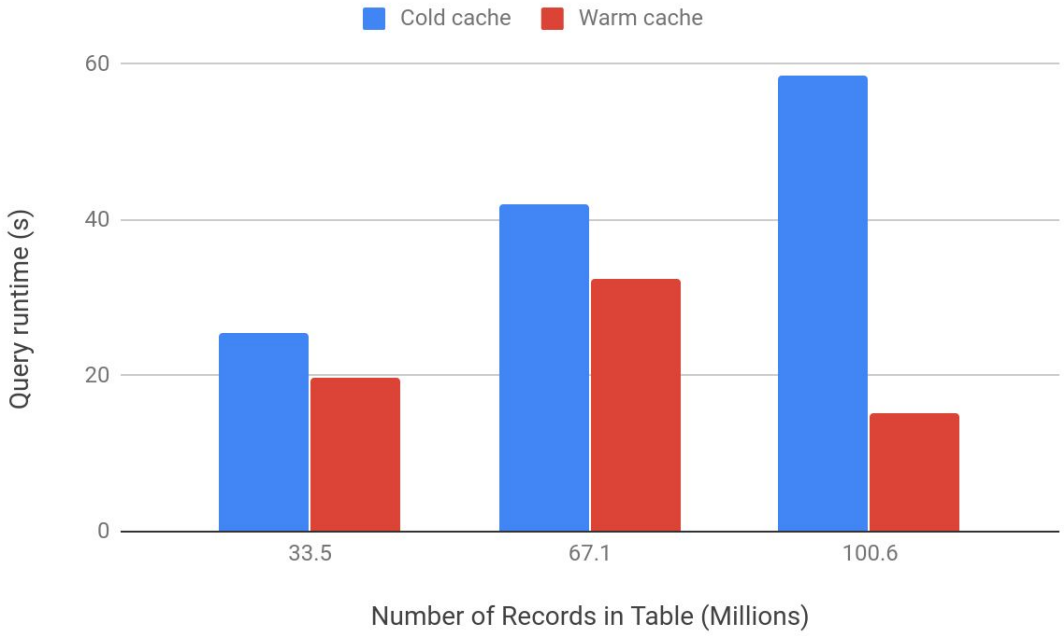

Figure 7: Query performance to filter all anomalous segments in different data table sizes

```
SELECT segment_start_ts AS start_date, COUNT(*) as number_points,
json_agg(value_to_passthru ORDER BY cur_ts) AS ecg_mv
FROM filter_segments(NULL::public.ecg_data_with_lag,
                'ecg_datetime', 'ecg_datetime_prev',
                'anomaly_likelihood', 'ecg_mv', 120, 423)
GROUP BY segment_start_ts
ORDER BY number_points DESC
LIMIT 10;
```

Figure 8: Query used for benchmark

We can see that the query times are quite long on cold cache especially. However, in practice queries will generally be run on a warm cache, a more restrictive date filter range would be used, and a more restrictive detector output range would be used. Additionally, faster storage media would typically be used (e.g., HDDs in RAID or SSDs) so this measurement represents a worst case scenario. On an NVMe laptop, we typically see execution times in the 1s range on the 33.5M data table. Also note while the runtimes are increasing with larger data sizes in the figure, applying a date filter so that the searched time scales are consistent between all data sizes (e.g., 1 year) causes runtimes to be comparable across all the data table sizes.

Finally, we compared the segment search on our TimescaleDB setup vs a stock PostgreSQL table, where the timestamp column is used as a primary key. We found that TimescaleDB performed 2X faster.

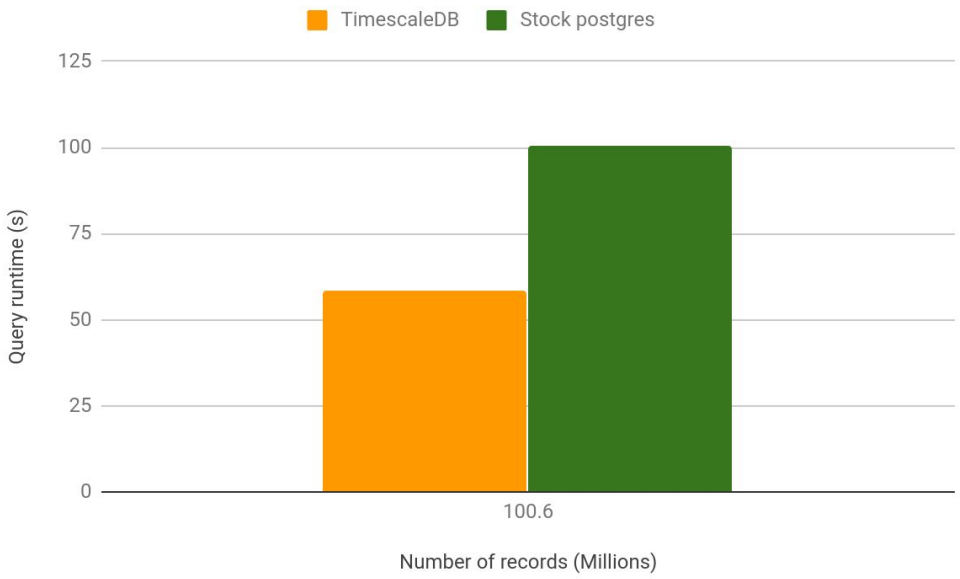

Figure 9: Segment filter search performance on TimescaleDB vs stock PostgreSQL

Future work could explore custom index structures to associate segment points with each other. Additionally, there is an optimization opportunity from converting the function into a c function because looping through the sorted filtered results to append segment start times takes about 65% of the function execution time and would be performed faster if it was not done in PL/pgsql.

Window Downsampling

After filtering segments and drilling down in the UI to explore the time-series data via pan/zoom, we ran into issues where the the interface would freeze or crash if the user zoomed out too much. This was because too much data was being sent to the frontend, forcing it to render a large amount of data. We only tested setups where the database and UI backend/frontend were on the same server, but in an actual implementation this would also fill up network bandwidth.

For this reason, we needed an operator that would automatically downsample data sent from the database based on a provided window size. We implemented an additional PL/pgSQL function to do this:

window_autosample(TABLENAME, datecol TEXT, valuecol_to_passthru TEXT, start_ts TIMESTAMP, end_ts TIMESTAMP, res_width INT, res_height INT)

This operator returns a filtered relation that contains sampled timestamps and values. Our current implementation is very basic and simply downsamples if the number of points required is larger than some multiple (currently 4) of the screen width. It then down samples by time bucketing to equal sized time buckets in the time range (to account for the case when time points may be non-uniformly sampled) then uses average values. Time bucketing is performed using TimescaleDB's custom time_bucket function, which is an optimized, more powerful version of PostgreSQL's standard *date_trunc* function.

```
Width_factor = 4

Max_samples = width_factor * screen_width

# Execute a count to get number of records in time range
Count = SELECT COUNT(*) from time range

If count <= max_samples:
   Return original relation without downsampling

Timespan_in_sec = date_max - date_min
Bucketsize_in_secs = int(
     timespan_in_sec / max_samples)

Return relation time bucketed and grouped by
bucketsize
```

Algorithm 2: Pseudocode for window_autosample

However, we realize that average values are not necessarily the most accurate way to downsample values, and there are many other techniques that could be used. A master's thesis by [Steinarsson, 2013] provides an overview of different techniques and proposes a technique inspired by cartography dubbed the "Largest-Triangle-Three-Buckets algorithm" for accurate time series downsampling that can serve as a basis for future improvements.

On the 100M synthetic ECG data set, we measured performance of the window_autosample operator across different filtered date ranges (1 hour, 1 day, 2 days, 3 days, 1 week, and 1 month), which are shown in the following figure along with the number of points in the unfiltered data set these periods correspond to.

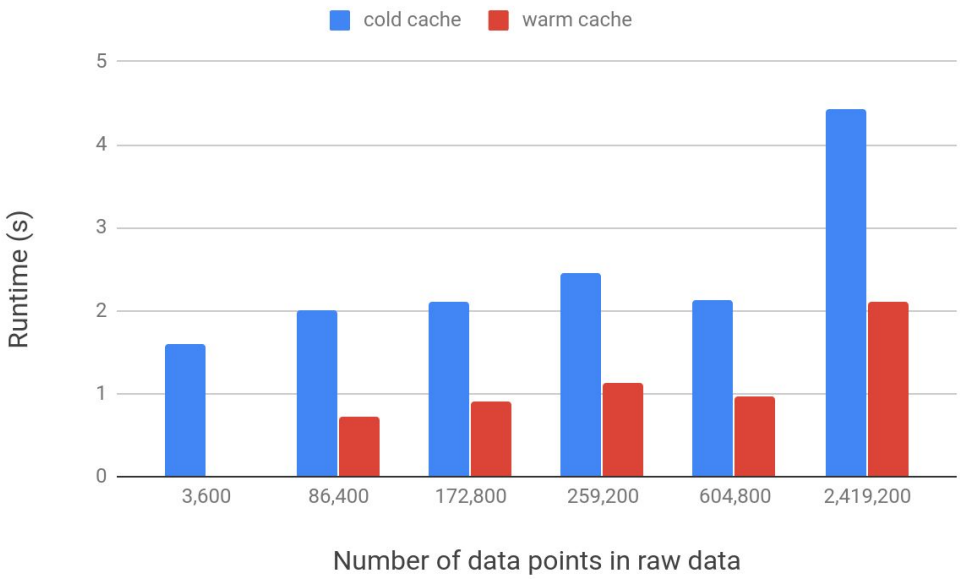

Figure 10: Window downsampling performance on different data sizes

```
SELECT * FROM window_autosample(NULL::ecg_data100,
'ecg_datetime',
'ecg_mv', '2018-02-01 12:00:00', '2018-02-01 13:00:00', 1000, 300);
```

Figure 11: Example window_autosample benchmark query

We can infer that the main performance bottleneck is the number of rows that must be scanned to downsample. A clever indexing structure or approximate query technique would probably be useful here to avoid having to do a full scan of the data within a time range.

Known Bugs

A known bug in the UI is that the time series isn't re-rendered correctly if one zooms out very far (e.g., a week), pans to a section of the time series several days away, and then tries to zoom in on that segment because the UI isn't triggered to resample at the updated higher zoom rate unless the largest zoom window you had used before has moved to a completely new window range. This is a minor bug that we did not get around to fixing before the deadline.

All experiments were performed on Postgres 10 because TimescaleDB hypertables in Postgres 11 performed on the order of 7-8x slower than the same queries running on Postgres 10. It was unclear whether Postgres 11 was still in beta support by Timescale. We think this is related to query parallelism and contention with hypertables (which are abstractions of many underlying PostgreSQL tables), but did not explore this further.

Additionally, we were hoping to write an update rule so that the table with lagging timestamps would be updated upon inserts to the main time series hypertable, but TimescaleDB does not allow update rules on hypertables.

Future Work

There are many further optimizations that can be added to the UI, including the features mentioned in the UI section and improved caching/rendering techniques.

On the data management side, besides improving the performance of the system through techniques such as column store and compression (which would likely require moving off TimescaleDB), we think there is opportunity to implement an operator similar in spirit to the Skyline operator to find “interesting segments that aren’t dominated by other segments”. The basic operator was for data mining applications generally [Borzsony et al., 2001], and work by later researchers experimented with time series data [Jiang and Pei, 2009], but there doesn’t seem to have been much work since then.

Code

Code is stored on github as a public repository: https://github.com/junjaytan/cs2270project

References

- S Borzsony, D Kossmann, K Stocker. The Skyline operator. Proceedings 17th International Conference on Data Engineering. 2001.
- P Eichmann, F Solleza, J Tan, N Tatbul, S Zdonik. Metro-Viz: Black-Box Analysis of Time Series Anomaly Detectors. CHI 2019.
- J Gao, P Agarwal, J Yang. Durable top-k queries on temporal data. PVLDB vol 11:13. Sep 2018.
- H Hochheiser, B Scneiderman. Timesearcher: Visual Exploration of Time-Series Data (multiple papers linked). 2004. http://www.cs.umd.edu/hcil/timesearcher/
- I Ilyas, G Beskales, M Soliman. A Survey of top-k query processing techniques in relational database systems. ACM Computing Surveys. Vol 4: 4. Oct 2008.
- B Jiang, J Pei. Online Interval Skyline Queries on Time Series. 2009 IEEE 25th International Conference on Data Engineering.
- AEW Johnson, TJ Pollard, L She, L Lehman, M Feng, M Ghassemi, B Moody, P Szolovits, LA Celi, RG Mark. MIMIC-III, a freely accessible critical care database. 2016. DOI: 10.1038/sdata.2016.35. https://mimic.physionet.org/
- E Keogh, J Lin, A Fu. Time Series Discords data sets. 2005. http://www.cs.ucr.edu/~eamonn/discords/
- F Li, K Yi, W Le. Top-k queries on temporal data. PVLDB vol 19:5. Oct 2010.
- PE McSharry, GD Clifford, L Tarassenko, L Smith. A dynamical model for generating synthetic electrocardiogram signals. IEEE Transactions on Biomedical Engineering 50(3): 289-294. March 2003. https://physionet.org/physiotools/ecgsyn/
- Numenta Anomaly Benchmark. https://github.com/numenta/NAB
- A Satyanarayan, D Moritz, K Wongsuphasawat, J Heer. Vega-Lite: A Grammar of Interactive Graphics. IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis), 2017.
- S Steinarsson. Downsampling Time Series for Visual Representation. University of Iceland. Masters Thesis. June 2013. https://skemman.is/bitstream/1946/15343/3/SS_MSthesis.pdf
- TimescaleDB website. https://www.timescale.com/
- W Tao, X Liu, Y Wang, L Battle, C Demiralp, R Chang, M Stonebraker. Kyrix: Interactive Pan/Zoom Visualizations at Scale. EuroVis 2019.